Тема

«Нововведения в java 7»

Состав

Project Coin

Пакет NIO2

Project Coin

Improved type inference for generic instance creation (diamond)

Это так называемый оператор *diamond (бриллиант, алмаз)*: **<>**.

В качестве примера часто приводят такой код:

```
// Java 7
        List<String> a = new ArrayList<>();
// до Java 7
        List<String> b= new ArrayList<String>();
```

Можно сделать еще проще? Вот так:

```
// в  Java 7
        List<String> a  = new ArrayList<>();
// до Java 7
        List<String> b = new ArrayList();
```

Или более “чистый” пример:

```
        List<String> a = new ArrayList<>();

        List<Integer> b = new ArrayList(a);
        List<Integer> c = new ArrayList<>(a);
```

Таким образом, компилятор автоматически выведет тип. Затем поймет, что у нас должен быть конструктор от списка чисел. Потом компилятор понимает, что нам вместо этого подают список строк и создаст ошибку компиляции.

try-with-resource

Рассмотрено в рамках занятия «Операторы и структура кода. Исключения»

String в switch-case выражениях

Пример:

```
String s = ...
switch(s) {
    case "foo": processFoo(s);
    break;
}
```

Целые числа в двоичном представлении и подчеркивание

Стало попроще работать с целыми числами.

Во-первых можно писать в двоичном виде:
int mask = 0b1000; // = 8
В итоге работать с битовыми масками стало намного удобней.

Пример из спецификации:

IntegerLiteral:
DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
BinaryIntegerLiteral

BinaryIntegerLiteral:
BinaryNumeral IntegerTypeSuffixopt

BinaryNumeral:
0 b BinaryDigits
0 B BinaryDigits

BinaryDigits:
BinaryDigit
BinaryDigit BinaryDigitsAndUnderscoresopt BinaryDigit

BinaryDigitsAndUnderscores:
BinaryDigitOrUnderscore
BinaryDigitsAndUnderscores BinaryDigitOrUnderscore

BinaryDigitOrUnderscore:
BinaryDigit
_

BinaryDigit: one of
0 1

Другими словами формат такой: **ноль** (0), символ '**b**' или '**B**', затем последовательность чисел из 1 и 0, также между ними можно использовать знак подчеркивания '_'.
Что касается подчеркивания, то это тоже новшество Java 7.

```
int a = 555_445_577;
int b = 0b1000_1111;

// а вот так, нельзя:
//   int c = _123; Не скомпилируется! error: illegal underscore
//   int d = 456_; Не скомпилируется! error: illegal underscore
```

Возможность ловить несколько разных исключений и более точная переброска

Интересная доработка с обработкой исключений:

```
private static void multiCatch() throws IOException {
     FileInputStream fis = new FileInputStream("/tmp/1.txt")
     try {
         fis.read();
     } catch (FileNotFoundException | SecurityException e) {
```

```
            throw e;
        } finally {
            fis.close();
        }
    }
```

Внешне это похоже на то ***как будто*** мы, через оператор '|' (*или*) указываем классы исключения, для которых стоит выполнять указанный код-обработки .
То есть в данном примере для FileNotFoundException и SecurityException.
Это позволяет в некоторых местах убрать дублирующийся код, а это уже хоть малая, но приятная победа над злом.

Второе новшество *more precise rethrow* - более интересное.

Рассмотрим такой пример кода:

```
private static void finalThrow() {
        try {
            throw new RuntimeException("test");
        } catch (final Exception e) {
            throw e;
        }
    }
```

Здесь в первую очередь сразу бросается в глаза **final** перед *Exception*!
Если бы мы попробовали скомпилировать такой код, в Java 6, то получили бы ошибку "... must be caught or declared to be thrown"!.

Здесь же, компилятор увидев **final** задумается, посмотрит на код и поймет, что на самом деле метод finalThrow() **не должен** кидать исключение Exception, и нет нужды в указании *throws Exception* для метода finalThrow().

Интрига на этом не заканчивается. Самое интересное то, что люди начали возмущаться! Зачем указывать **final**?!
Это дополнительный (лишний) смысл к этому ключевому слову!
В итоге, поведение без final **по-умолчанию** (если это возможно) аналогично тому, как если бы мы указали **final**.

Например:

```
    private static void finalThrow() {
        try {
            throw new RuntimeException("test");
        } catch ( /* final */ Exception e) {
            // final можно не указывать.
            throw e;
        }
    }

   // НЕ СКОМПИЛИРУЕТСЯ!!!!!
    private static void finalThrow2() {
        try {
            throw new RuntimeException("test");
        } catch (Exception e) {
            // поломали. e - не final!!!!!!
            e = e;
            throw e;
        }
    }
```

Упрощенный вызов методов с переменным количеством аргументов

Это улучшение больше похоже на хак.
Сделали новую аннотацию: SafeVarargs
Это аннотация глушит предупреждения, которые мог бы выкидывать компилятор при использовании дженериков вместе с вараргами.

Пакет NIO2

Это совершенно новый подход, в отличии от старого java.io.File призванный полностью его заменить во всех аспектах, касающихся взаимодействия с файловой системой.

NIO2 уже по-умолчанию имеет удобные возможности исполнения в многопоточном приложении. Без мучительных конфигураций можно исполнять операции работы с фаловой системой или сетью в фоновом потоке.

Со всех сторон упрощает кодинг и привносит даже новые возможности? которые ранее были не доступны. Например работа напрямую с символическими ссылками.
Вот некоторые из методов, которые сильно упрощают жизнь:

Files.walkFileTree()
Files.isSymbolicLink()
Files.readAttributes()

Новое апи хорошо оптимизировали для работы с конкретной ОС используюся ее те или иные нативные преимущества для скорости работы с ФС сетью, большими файлами.

Улучшения касаются оптимизации работы приложения на нескольких многоядерных процессоров, которые позволяют программе исполняться в "настоящей" мультипоточной манере. Представляет программистам достаточно простую абстракцию для мультипоточной работы с файлами и сокетами.

Рассмотрим более бодробно возможности апи
Как я уже писл ранее класс Path это фундамент для работы с файловым вводом выводом не зависимо от операционной системы. И может представляться например следующим видом "c:/user/dir" или "/user/dir" (*nix os).

Path это абстрактная конструкция. Объект можно создать и даже работать с ним, но он может физически не относиться не к одному файлу в сисетеме, пока мы его не создадим очевидным образом - Files.createFile(Path target). Иначе если будем писать или читать до создания то получим IOException. Тоже самое случиться если файл не существует, а я его попытаю просчитать. Кстати говолря, JVM привязывает объекты к физическим только в рантайме.

Апи nio2 разделяет расположение (Path) и манипуляцию с файл системой (класс File)

Path не ограничевается представлением классических ФС. Класс так же может представлять местонахождения файлов и папок в архивах например jar или zip.

Описание основных ключевых классов:
Path включает методы для получения информации о пути до расположения файлы или папки. Для доступа к элементам пути. Для конвертации в друге формы. Или извлечения части пути. Еще есть методы для распознания строки., представляющей путь и для удаления ибыточности из пути (многословности).
Paths класс помощник включающий вспомогательные методы для формирования пути например, get(String first, String... more) или get(URI uri).
FileSystem класс интерфейс к файловой системе.
FileSystems вспомогательный класс с хелперами.

Создание Path:
Path listing = Paths.get("/usr/bin/zip");
или эквивалент:
Path listing = FileSystems.getDefault().getPath("/usr/bin/zip");

Получение связанной информации:
Имя файла listing.getFileName() = zip
Количество именованных элементов в пути listing.getNameCount() = 3
Родительский каталог listing.getParent() = bin
Корень ФС listing.getRoot() = /
Количество подкаталогов от корня до файла (глубина) listing.subpath(0, 2) = 2

Оптимизация путей:
Можно привести ссылку к реальному файло на который она ссылается так
Path realPath = Paths.get("/usr/logs/log1.txt").toRealPath();

Можно соединить две папки:
Path prefix = Paths.get("/uat/");
Path completePath = prefix.resolve("conf/application.properties");

Нет никаких проблем при работе с легаси кодом их старого пакета java.io.File который естественно в 7 джаве тоже есть для обратной совместимости.
Для этого есть такие методы:
toPath() и toFile() , где первый переводит к современному Path, а второй к старому File.

Работа с файловой системой
С точки зрения апи файлы и папки рассматриваются как единые сущности различающиеся лишь соответствующими атрибутами.
nio2 предоставляет слудующие как стандартные, так и специфические этому апи возмозможности

Создание и удаление файлов

Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Path file = Files.createFile(target);

можно задать атрибут FileAttribute в примере на чтение и запись всем. для posix систем (например UNIX частично Windows итд)
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr = PosixFilePermissions.asFileAttribute(perms);
Files.createFile(target, attr);

Для удаления файлов нужно иметь необходимые привилегии для пользователя из под которого запущен процесс

Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.delete(target);

Копирование и перемещение

Используя вспомогательные методы из класса File можно копировать и перемещать файлы с различными вариантами опций.

Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.**copy**(source, target, REPLACE_EXISTING);

Есть много разных опций. Например ATOMIC_MOVE - операция перемещения завершается

успехом если обе стороны процесса успешны (удалено вставлено в новое место).

```
Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.move(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

Чтение и запись атрибутов

```
Path zip = Paths.get("/usr/bin/zip");
Files.getLastModifiedTime(zip);
Files.size(zip);
Files.isSymbolicLink(zip);

Files.isDirectory(zip);
```

Это стандартные атрибуты поддерживаемые большинством файловых систем, так же можно читать атрибуты присущие определенным ФС например posix:

```
Path profile = Paths.get("/user/Admin/.profile");
PosixFileAttributes attrs =Files.readAttributes(profile,PosixFileAttributes.class);
Set<PosixFilePermission> posixPermissions = attrs.permissions();
posixPermissions.clear();
posixPermissions.add(GROUP_READ);
...
Files.setPosixFilePermissions(profile, posixPermissions);
```

Чтение и запись в файл
Апи позволяет исать и читать файлы открывая их напрямую для буферезированных reader writer и читать их построчно. Для обратной совместимостью Path так же работает с обычными (старыми) input output потоками.

Чтение

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedReader reader = Files.newBufferedReader(logFile, StandardCharsets.UTF_8)) {
String line;
while ((line = reader.readLine()) != null) {
...
}
}
```

Запись

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedWriter writer = Files.newBufferedWrite(logFile, StandardCharsets.UTF_8,
StandardOpenOption.WRITE))
{
   writer.write("Hello World!");
   ..
}
```

StandardOpenOption.WRITE это один из возможных аргументов для открытия файла. Аргумент типа varargs.

Для обратной совместимости со старым апи надо использовать методы обертки для nio в классе Files:

```
Files.newInputStream(Path, OpenOption...)
Files.newOutputStream(Path,OpenOption...)
```

В некоторых случаях для удобства чтения можно воспользоваться дополнительными методами:

```
Path logFile = Paths.get("/tmp/app.log");
List<String> lines = Files.readAllLines(logFile, StandardCharsets.UTF_8);
byte[] bytes = Files.readAllBytes(logFile);
```

Сервис отслеживания изменений WatchService

Для удобного мониторинга изменения файлов и атрибутов добавлен новый класс java.nio .file.WatchService.

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = FileSystems.getDefault().getPath("/usr/karianna");
WatchKey key = dir.register(watcher, ENTRY_MODIFY);
while(!shutdown) {
  key = watcher.take();
  for (WatchEvent<?> event: key.pollEvents()) {
    if (event.kind() == ENTRY_MODIFY){
      System.out.println("Home dir changed!");
    }
  }
key.reset();
}
```

Код работает в бесконечном цикле (флаг shutdown). watcher.take() берем снимок прошедших событий (первый раз конечно изменений не будет поэтому для постоянного мониторинга цикл и нужен) проходимся по коллекции ищем интересующее нас событие - если есть? значит произошло и было зарегистрировано. key.reset(); сбрасываем состояние. Один из вариантов рефакторинга реализовать патерн observer. Объекты смогут работать по методу подписки на событие.

SeekableByteChannel улучшенный bytechannel

В java 7 вводится новый интерфейс SeekableByteChannel предоставляющий удобную возможность разработчику изменения позиции курсора (отслеживания) и размера установки размера буфера для считывания данных из канала

```
Path logFile = Paths.get("c:\\temp.log");
ByteBuffer buffer = ByteBuffer.allocate(1024);
FileChannel channel = FileChannel.open(logFile, StandardOpenOption.READ);
channel.read(buffer, channel.size() - 1000);
```

Материалы для подготовки

1. http://amaloff.blogspot.ru/2015/09/nio2-java-io.html
2. https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html
3. https://www.javaworld.com/article/2882984/learn-java/nio2-cookbook-part-1.html

Вопросы для самоконтроля

1) В чем проявляется типобезопасность Diamond-оператора в java?
2) Для чего нужен WatchService?
3) В чем состоит новшество в NIO2 в классе Path?